

## A FAST ITERATIVE METHOD FOR EIKONAL EQUATIONS\*

WON-KI JEONG<sup>†</sup> AND ROSS T. WHITAKER<sup>†</sup>

**Abstract.** In this paper we propose a novel computational technique to solve the Eikonal equation efficiently on parallel architectures. The proposed method manages the list of active nodes and iteratively updates the solutions on those nodes until they converge. Nodes are added to or removed from the list based on a convergence measure, but the management of this list does not entail an extra burden of expensive ordered data structures or special updating sequences. The proposed method has suboptimal worst-case performance but, in practice, on real and synthetic datasets, runs faster than guaranteed-optimal alternatives. Furthermore, the proposed method uses only local, synchronous updates and therefore has better cache coherency, is simple to implement, and scales efficiently on parallel architectures. This paper describes the method, proves its consistency, gives a performance analysis that compares the proposed method against the state-of-the-art Eikonal solvers, and describes the implementation on a single instruction multiple datastream (SIMD) parallel architecture.

**Key words.** Hamilton–Jacobi equation, Eikonal equation, viscosity solution, label-correcting method, parallel algorithm, graphics processing unit (GPU)

**AMS subject classifications.** 70H20, 49L25, 68W10, 05C12

**DOI.** 10.1137/060670298

**1. Introduction.** Applications of the Eikonal equation are numerous. The equation arises in the fields of computer vision, image processing, computer graphics, geoscience, and medical image analysis. For example, the shape-from-shading problem, which infers three-dimensional (3D) surface shape from the intensity values in two-dimensional (2D) images, can be described as a solution to the Eikonal equation [6, 28]. Extracting the medial axis or skeleton of shapes is often done by analyzing solutions of the Eikonal equation with the boundary conditions specified at the shape contour [34]. Solutions to the Eikonal equation have been proposed for noise removal, feature detection, and segmentation [21, 31]. In computer graphics, geodesic distance on discrete and parametric surfaces can be computed by solving Eikonal equations defined on discrete surfaces or the parametric domain using the gradients on smooth surfaces [22, 19, 36]. In physics, the Eikonal equation arises in models of wavefront propagation. For instance, the calculation of the travel times of the optimal trajectories of seismic waves is a critical process for seismic tomography [26, 33]. Figure 1 shows an example of 3D seismic wave propagation using the solution of the Eikonal equation, where the isosurfaces of different arrival times are rendered using different colors, and the colored slice is a cross section of the speed volume in which the different colors represent different speed values.

With the advent of recent technical developments in multicore and massively parallel processors, computationally expensive scientific problems are now feasible on consumer-level PCs; these same problems were feasible only on expensive large-scale multiprocessor machines or cluster systems in the very recent past. For example, the

\*Received by the editors September 19, 2006; accepted for publication (in revised form) April 28, 2008; published electronically July 23, 2008. This work was supported in part by the ExxonMobil Upstream Research Company and an NVIDIA fellowship.

<http://www.siam.org/journals/sisc/30-5/67029.html>

<sup>†</sup>Scientific Computing and Imaging Institute, School of Computing, University of Utah, 72 S. Central Campus Drive, 3750 WEB, Salt Lake City, UT 84112 (wkjeong@cs.utah.edu, whitaker@cs.utah.edu).

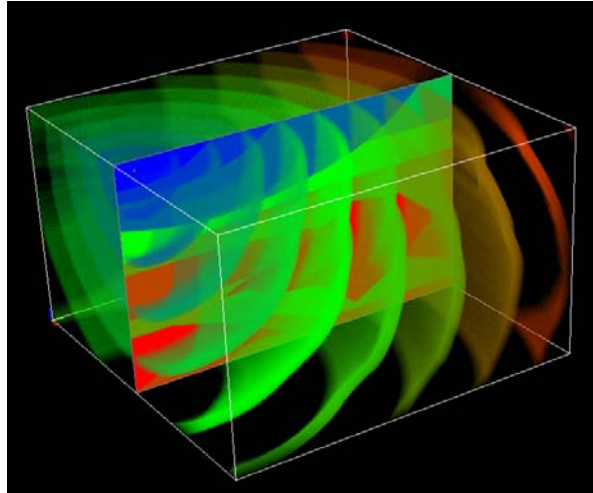


FIG. 1. Wave propagation simulation on a 3D synthetic seismic speed volume.

most recent graphics processors, which cost only several hundred dollars, can reach a peak performance of up to several hundred giga floating point operations per second (GFLOPS); that was the performance of a supercomputer a decade ago. Among the many parallel architectures, researchers are interested in single instruction multiple datastream (*SIMD*) *streaming architectures*, for example, Imagine [17], Cell [8], and graphics processors [7], for high performance computing problems these days. Modern SIMD streaming architectures provide massively parallel computing units (up to several hundred cores) with rather simple branching circuits and a huge memory bandwidth. This simple architecture enables computing-intensive local operations, i.e., kernels, over a large stream of data very efficiently, and therefore many data-parallel problems, e.g., multimedia data processing, computer graphics, and scientific computation, map very well on such simple and scalable architectures. The Eikonal equation solver can be one such computing-intensive application for SIMD streaming architecture because even the fastest CPU-based Eikonal solver is still slow on reasonably large 3D datasets. However, it is not straightforward to implement existing Eikonal solvers on parallel architectures because most of the methods are designed for a serial processing model, and therefore it will be interesting to develop fast parallel algorithms to solve the Eikonal equation on SIMD streaming architectures.

The most popular Eikonal equation solvers [14] are based on heterogeneous data structures and irregular data updating schemes, which hinder implementing the methods on parallel architectures. For instance, the fast marching method (FMM) [29, 30] uses a heap data structure, which includes grid points from the entire wavefront. During each iteration, the heap determines the as yet unsolved grid value that is guaranteed to depend only on neighbors whose values are fixed (i.e., the solution on a grid point is computed and the grid point is removed from the heap). The heap must be updated whenever each grid value is replaced by a new solution, which cannot be done in parallel. For such a serial algorithm, the heap becomes a bottleneck that does not allow for massively parallel solutions, such as those available with SIMD architectures. Furthermore, grid points must be updated one at a time, in a way that does not guarantee locality, limiting opportunities for coherency in cache or local memory. Another popular method, the fast sweeping method (FSM) [40], uses an alternating

Gauss–Seidel update to speed up the convergence because the method does not rely on a sorting data structure as FMM does. A Gauss–Seidel update requires reading from and writing to the single memory location, and it is inefficient or prohibited on some of the most efficient parallel architectures. Furthermore, as we shall see in the results of this paper, the performance of FSM highly depends on the complexity of input data. Thus, there remains a need for fast Eikonal solvers that can run efficiently on *both* single processors and parallel SIMD architectures for complex data sets.

In this paper we propose a new computational technique to solve the Eikonal equation efficiently on parallel architectures; we call it the fast iterative method (FIM). The proposed method relies on a modification of a *label-correcting method*, which is a well-known shortest path algorithm for graphs, for efficient mapping to parallel SIMD processors. Unlike the traditional label-correcting methods, the proposed method employs a simple list management method based on a convergence measure and selectively updates multiple points in the list simultaneously using a Jacobi update for parallelization. The proposed method is simple to implement and faster than the existing Eikonal equation solvers on a wide range of speed functions on both single and parallel processors. FIM is an algorithmic framework to solve the Eikonal equation independent of numerical schemes, for example, the finite difference method using a Godunov Hamiltonian or a finite element method using optimal control theory, and therefore can be extended easily to solve a wide class of boundary value problems including general Hamilton–Jacobi equations [31].

The main contribution of this paper is twofold. First, we introduce the FIM algorithm and perform a careful empirical analysis on a single processor in order to make direct comparisons against other CPU-based methods to understand the benefits and limitations of each method. These empirical comparisons shed light on the theoretical worst-case claims of the earlier work. We show that although the worst-case performance of the proposed algorithm is not optimal, it performs much better than worst-case on a variety of complex data sets. While several papers argue [18, 40] that the complexity of their algorithms *is as low as*  $O(N)$  (i.e., best case), there have been only a few systematic studies of how often such methods achieve these best cases or what the constants are that affect the complexity [14]. The actual performance of these algorithms is affected by many different factors, including the size of the datasets and the geometric configurations of the speed functions. Second, we propose a block-based FIM specifically for more efficient implementation of the proposed method on massively parallel SIMD architectures. We present the experimental result of the block-based FIM and compare it to the result of the CPU-based methods to elaborate how the proposed method scales well on SIMD parallel architectures in a realistic setup.

The remainder of this paper proceeds as follows. In the next section we give formal problem definitions and notation used in this article. In section 3, we introduce previous work on the Eikonal equation solvers in detail to elucidate the advantages and disadvantages of existing methods. In section 4 we introduce the proposed FIM and its variant for SIMD parallel architecture in detail. In section 5 we show numerical results on a number of different examples and compare them with the existing methods. In section 6 we summarize the paper and discuss future research directions related to this work.

**2. Notation and definitions.** In this paper, we consider the numerical solution of the Eikonal equation (1), a special case of nonlinear Hamilton–Jacobi partial

differential equations (PDEs), defined on a Cartesian grid with a scalar speed function

$$(1) \quad H(\mathbf{x}, \nabla \phi) = |\nabla \phi(\mathbf{x})|^2 - \frac{1}{f^2(\mathbf{x})} = 0 \quad \forall \mathbf{x} \in \Omega,$$

where  $\Omega$  is a domain in  $R^n$ ,  $\phi(\mathbf{x})$  is the travel time or distance from the source, and  $f(\mathbf{x})$  is a positive speed function defined on  $\Omega$ . The solution of the Eikonal equation with an arbitrary speed function is sometimes referred to as a *weighted distance* [36] as opposed to a Euclidean distance for a constant speed function.

We refer to a *node* as a grid point on a Cartesian grid where the discrete samples or solutions of the continuous functions are defined. A node is usually defined as an  $n$ -tuple of numbers, for example,  $\mathbf{x} = (i, j, k)$  for a 3D case. We define an *edge* as a line segment that directly connects two nodes whose length defines a grid length  $h_p$  along the corresponding axis  $p \in \{x, y, z\}$ . We define an *adjacent neighbor* as a node connected by a single edge. For example, a node  $\mathbf{y} = (i + h_x, j, k)$  is the adjacent neighbor of the node  $\mathbf{x} = (i, j, k)$  along the positive  $x$  direction.

To compute a solution of (1), we use a Godunov upwind difference scheme as proposed in [28, 30, 40]. For example, the first order Godunov upwind discretization  $g(\mathbf{x})$  of the Hamiltonian  $H(\mathbf{x}, \nabla \phi)$  shown in (1) on a 3D Cartesian grid can be defined as follows:

$$(2) \quad g(\mathbf{x}) = \left[ \frac{(U(\mathbf{x}) - U(\mathbf{x})^{x\min})^+}{h_x} \right]^2 + \left[ \frac{(U(\mathbf{x}) - U(\mathbf{x})^{y\min})^+}{h_y} \right]^2 + \left[ \frac{(U(\mathbf{x}) - U(\mathbf{x})^{z\min})^+}{h_z} \right]^2 - \frac{1}{f(\mathbf{x})^2},$$

where  $U(\mathbf{x})$  is the discrete approximation to  $\phi$  at the node  $\mathbf{x} = (i, j, k)$ ,  $U(\mathbf{x})^{p\min}$  is the minimum  $U$  value among two adjacent neighbors of  $U(\mathbf{x})$  along the axis  $p \in \{x, y, z\}$  directions, respectively,  $h_p$  is the grid space along the axis  $p \in \{x, y, z\}$  direction, respectively, and  $(n)^+ = \max(n, 0)$ . We can solve (2) on a grid of any dimension, but in this paper, we focus our discussion on the Eikonal equation defined on 3D grids (volumes) commonly used in image processing applications. We refer to *upwind* neighbors of a node as the adjacent neighbors whose solutions have values less than or equal to the node in question. In the discussions that follow, *updating a node* refers to the process of computing a new solution on the node and replacing a previous solution on the node so long as the upwind (causal) relationship is satisfied. We say a node has *converged* when the difference between the solutions of the previous and the current update iterations is within a user-defined threshold.

**3. Previous work.** A number of different numerical strategies have been proposed to efficiently solve the Eikonal equation. Iterative schemes, for example [28], rely on a fixed-point method that solves a quadratic equation at each grid point in a predefined update order and repeats this process until the solutions on the whole grid converge. A drawback of this strategy is that it may converge slowly and the complexity (worst case) behaves as  $O(N^2)$ , where  $N$  is the number of nodes on the grid.

In some cases, the characteristic path, which is the optimal trajectory from one point to another, does not change its direction, as is the case in computing the shortest path on an homogeneous media. In such cases, we can solve the Eikonal equation by updating solutions along a specific direction without explicit checks for causality.

Based on this observation, Zhao [40] and Tsai et al. [38] proposed the (*FSM*), in which the Eikonal equation is solved on an  $n$ -dimensional grid using at least  $2^n$  directional sweeps, one per each quadrant, within a Gauss–Seidel update scheme, where a raster-scan sweeping idea was originally proposed by Danielsson [9] to compute the Euclidean distance on a binary image. Figure 2 shows a 2D example of a quadrant and the neighbor dependency that can be solved using a single sweep. The method may converge faster than a Jacobi update method, which updates all grid points at once, but the algorithm complexity ( $O(kN)$ ) still depends on the complexity of the input speed function, and our experiments show that in most cases FSM is not particularly efficient (relative to other methods) except in very simple cases, such as a single source point with a constant speed function.

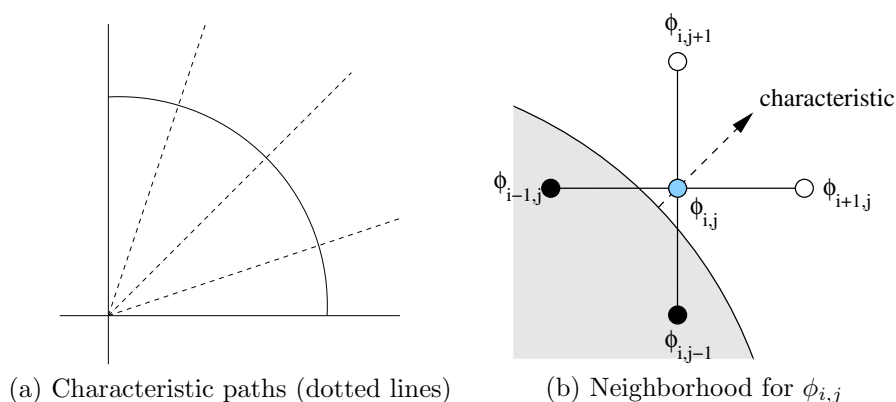


FIG. 2. 2D example of characteristic paths and neighbors for solving the Eikonal equation in the first quadrant.

Some adaptive, iterative methods based on a *label-correcting algorithm* (from a similar shortest-path problem on graphs [2]) have been proposed [24, 3, 10, 11]. The methods are based on two strategies: avoiding expensive sorting data structures and allowing multiple updates per node point. Falcone [10, 11] proposed using flags for adaptive updating of grid points. These algorithms compute upwind neighbors (and solutions) from a tentative solution on the grid and keep track of which nodes are potentially *out of date* with the current solution, and such nodes are considered *active* and require further updates. The status of nodes (*up to date* or *out of date*) is modified to reflect the causal relationships in the Eikonal equation. Polymenakos, Bertsekas, and Tsitsiklis [24] proposed using a simple queue to manage the active grid points and solved a Hamilton–Jacobi–Bellman equation. Recently Bornemann and Rasch [3] proposed an adaptive Gauss–Seidel update to solve a finite element discretization of the Hamilton–Jacobi equation. Those label-correcting-type methods share some common ideas with the proposed method, but they do not scale well on the parallel architectures because most of the methods use a single queue for a serial update or a predefined updating sequence, which impairs the efficiency of SIMD parallel processing or memory coherency. In addition, all the above methods employ a finite element method based on the optimal control theory to solve the Hamilton–Jacobi equation, which entails the burden of solving an optimization problem (even though the neighborhood is known), and therefore it is more expensive than a closed form solution of a Godunov discretization of the Hamiltonian used in the proposed method.

The *expanding wavefront scheme* allows the wavefront to expand *in the order of the causality* given by the equation and the speed function. An early work of Qin et al. [25] introduces this approach, but they use a brute-force-type sorting method that impairs the performance. Tsitsiklis [39] proposed the use of a sorted list of active nodes and thereby describes the first optimal algorithm for the Eikonal equation. A version of this algorithm is called the *fast marching method (FMM)* by Sethian [29] and Sethian and Vladimirsky [30, 32], which is currently the mostly widely used expanding wavefront method. FMM relies on a heap data structure to keep the updating sequence, and the complexity is  $O(N \log N)$ . The performance of the algorithm depends on the size of the input data and the data structure. Inserting or deleting an element in a heap requires  $O(\log h)$ , where  $h$  is the size of the wavefront. This is done for each point on the grid, which gives  $O(N \log h)$ , and because  $h < N$  we have  $O(N \log N)$  as the asymptotic worst case, which is optimal. As we might expect, the algorithm performs consistently, and variations in performance depend only on the complexity of the wavefront, which changes the average values of  $h$ . In practice, however, the  $\log h$  cost of maintaining the heap can be quite significant, especially for large grids or when solving for wavefronts in three or more dimensions.

Kim [18] proposed a group updating method based on a global bound without using a sorting data structure, called *group marching method (GMM)*. GMM avoids using a heap but maintains a list of points using a simple data structure, e.g., a linked list, and updates a group of points that satisfies the condition given in the paper at the same time. The author argues that the computational complexity of the proposed method is, in principle,  $O(N)$ . However, the method requires a knowledge of the minimum  $\phi$  and  $f$  in the current nodes in the list to determine the group in every update step. In general, it may require  $O(N)$  to find this value. To resolve this problem, Kim suggests using a global bound which increases incrementally as the wavefront propagates where the increment is determined in the preprocessing stage by the input speed function. If the speed function has large contrasts, then only a few points (or even no points) can be updated at each iteration, and the algorithm may require a large number of iterations, which impairs performance.

Even though there has been much effort to develop parallel algorithms for general purpose computing problems, only a few studies have been done to solve a general Eikonal equation with arbitrary speed functions on 2D or 3D Cartesian grids, i.e., weighted distances on images (2D) and volumes (3D), on SIMD parallel streaming architectures. Most of the existing parallel methods, specifically on the GPU, focus on Euclidean distance transforms or Voronoi diagrams, which is equivalent to computing a solution of the Eikonal equation defined over a constant speed field. Hoff et al. [15] first proposed using OpenGL API and the hardware rasterization function in the fixed graphics pipeline to compute distance maps. The method approximates the distance function using polygonal meshes, i.e., cones, and uses a hardware rasterization to compute per-pixel distances. Sud, Otaduy, and Manocha [37] extended Hoff's method using culling and clamping algorithms to reduce unnecessary distance computations. Sigg, Peikert, and Gross [35] computed a signed distance field in a thin band around the triangular mesh using a fragment program and hardware rasterization. Distances from each primitive, e.g., face, edge, and vertex, are computed using linear interpolation inside a prism. Rong and Tan [27] proposed a *jump flooding* technique to compute approximated Voronoi diagram on the GPU. Fischer and Gotsman [13] proposed a GPU method to compute a high order Voronoi diagram using the tangent plane algorithm and a depth peeling technique. All the GPU methods shown

above are *not* applicable to the problem we address in this paper because they are not solving the Eikonal equation for arbitrary speed functions. Recent work by Bronstein et al. [4, 5] proposed a novel numerical scheme to solve tensor-based Eikonal equations to compute the geodesic distance on manifolds, which is an important problem that poses some computational challenges for the applications that must solve 2D and 3D equations at interactive rates. Bronstein et al. focus on the numerical side, i.e., a novel updating scheme based on splitting the obtuse angles of simplices to solve anisotropic Eikonal equations with tensor speed functions, but this paper focuses on the algorithmic side, i.e., a novel parallel algorithm for SIMD architectures to efficiently compute solutions on Cartesian grids using any existing updating schemes, such as Godunov upwind schemes [38]. Recently, a GPU-based Hamilton–Jacobi equation solver to analyze the connectivity in 3D DT-MRI volumes was proposed by the same authors of this paper [16]. However, in this paper, we give more in-depth analysis and theoretical discussion of the proposed algorithm as well as an empirical study and comparison to other existing methods on both serial and parallel architectures, which were not addressed in [16].

**4. Fast iterative method (FIM).** In this section we propose an Eikonal solver based on a selective iterative method, which we call the *fast iterative method (FIM)*. The main design goals in order to produce good overall performance, cache coherence, and scalability across multiple processors are:

1. the algorithm should not impose a particular update sequence,
2. the algorithm should not use a separate, heterogeneous data structure for sorting, and
3. the algorithm should be able to simultaneously update multiple points.

Criterion 1 is required for cache coherency and streaming architectures. For example, FMM updates solutions based on causality and requires random access to the memory which impairs cache coherency. Streaming architectures, such as GPUs, are optimized for certain memory access patterns, and therefore special update sequences given by the program could force one to violate those patterns. Control of the update sequence allows an algorithm to update whatever data is in cache (e.g., local on the grid). Criterion 2 is required for SIMD and streaming architectures. For example, FMM manages a heap to keep track of causality in the active list. Operations on the heap require multiple instructions on significantly smaller datasets (relative to the original grid), and this cannot be efficiently implemented on SIMD or streaming architecture because such machines are efficient only when they process large datasets with a common operation. Criterion 3 allows an algorithm to fully utilize the parallel architecture of cluster systems or SIMD processors to increase throughput.

**4.1. Algorithm description.** FIM is based on observations from two well-known numerical Eikonal solvers. One approach is the iterative method proposed by Rouy and Tourin [28], which updates every grid node iteratively until it converges. This method does not rely on the causality principle and is simple to implement. However, the method is inefficient because every grid node must be visited (and a quadratic is evaluated) until the solutions on the entire grid have converged. The other related approach is FMM [29, 30], which uses the idea of a narrow band of points on the wavefront, and thereby updates points selectively (one at a time) by managing a heap.

The main idea of FIM is to solve the Eikonal equation selectively on grid nodes without maintaining expensive data structures. FIM maintains a narrow band, called the *active list*, for storing the grid nodes that are being updated. Instead of using

a special data structure to keep track of exact causal relationships, we maintain a looser relationship and update all nodes in the active list simultaneously using parallel architecture. During each iteration, we expand the list of active nodes, and the band thickens or expands to include all nodes that could be affected by the current updates. A node can be removed from the active list when its solution is up to date with respect to its neighbors (i.e., it has converged) and can be appended to the list (relisted) when any upwind neighbor's value is changed. Because the proposed method uses a simple data structure and allows multiple updates per node, one may think of FIM as a variant of a *label-correcting algorithm* for the shortest graph path problems [2]. The pseudocode for a typical shortest-path algorithm is given in Algorithm 4.1.

---

ALGORITHM 4.1. GENERAL LABEL-CORRECTING ALGORITHM( $L$ ).

---

**comment:**  $L$  : Queue

**while**  $L$  is not empty

**do**  $\left\{ \begin{array}{l} \text{for } \mathbf{x} \in L \\ \quad \text{do} \left\{ \begin{array}{l} \text{remove } \mathbf{x} \text{ from } L \\ \quad \text{for adjacent neighbor } \mathbf{y} \text{ of } \mathbf{x} \\ \quad \quad \text{do} \left\{ \begin{array}{l} \text{update } \mathbf{y} \\ \quad \text{if } \mathbf{y} \text{ is not in } L \\ \quad \quad \text{then add } \mathbf{y} \text{ to } L \end{array} \right. \end{array} \right. \end{array} \right.$

---

Thus the conventional label-correcting algorithm is serial, meaning that only one node is removed from the list on any given iteration step. In addition, the top node of the queue is always removed on each iteration. Many existing label-correcting methods employ acceleration techniques that depend on the *serial* nature of the algorithm. For example, smallest label first (SLF) and last label last (LLL) methods in [24], or adaptive Gauss–Seidel update in [3], do not easily parallelize. Some parallel algorithms for graph shortest-path problems [1, 12] use multiple queues or split domain techniques for parallel implementation, but these strategies are not efficient on SIMD architectures. Therefore, we use a Jacobi update scheme to update all the nodes in the active list in parallel. In addition, because we update the whole wavefront on every iteration, we remove only the converged nodes from the list to minimize the growth of the active list.

The FIM algorithm consists of two parts, the initialization and the updating. In the initialization step, we set the boundary conditions on the grid, and set the values of the rest of the grid nodes to infinity (or some very large value). Next, the adjacent neighbors of the source nodes in the cardinal directions (4 in two dimensions and 6 in three dimensions) are added to the active list  $L$ . In the updating step, for every point  $\mathbf{x} \in L$  we compute the new  $U(\mathbf{x})$  by solving (2) and check if the node is converged by comparing the previous  $U(\mathbf{x})$  and the new solution on  $\mathbf{x}$ . If it is converged, we remove the node  $\mathbf{x}$  from  $L$  and add any nonconverged adjacent nodes to  $L$  if they are not in  $L$ . Note that newly added nodes must be updated in the successive iteration to ensure a correct Jacobi update. The updating step is repeated until  $L$  is empty. The pseudocode description of the proposed FIM algorithm is given (see Algorithm 4.2).

To compute a new solution  $g(\mathbf{x})$  of a node  $\mathbf{x}$  in the active list, we use the Godunov upwind discretization of the Eikonal equation (2) and solve the associated quadratic equation in closed form. Algorithm 4.3 is the pseudocode to compute the solution of the 3D Eikonal equation as introduced in [40], which takes as input values  $a, b, c$



of upwind neighbors in the cardinal directions of the grid and  $f$  as a positive scalar speed.

---

ALGORITHM 4.2. FIM( $X, L$ ).

---

**comment:** 1. Initialization ( $X$  : a set of grid nodes,  $L$  : active list)

**for each**  $\mathbf{x} \in X$

**do**  $\begin{cases} \text{if } \mathbf{x} \text{ is source node} \\ \text{then } U(\mathbf{x}) \leftarrow 0 \\ \text{else } U(\mathbf{x}) \leftarrow \infty \end{cases}$

**for each**  $\mathbf{x} \in X$

**do**  $\begin{cases} \text{if any neighbor of } \mathbf{x} \text{ is source} \\ \text{then add } \mathbf{x} \text{ to } L \end{cases}$

**comment:** 2. Update points in  $L$

**while**  $L$  is not empty

**do**  $\begin{cases} \text{for each } \mathbf{x} \in L \text{ in parallel} \\ \begin{cases} p \leftarrow U(\mathbf{x}) \\ q \leftarrow \text{solution of } g(\mathbf{x}) = 0 \\ U(\mathbf{x}) \leftarrow q \\ \text{if } |p - q| < \epsilon \\ \text{then } \begin{cases} \text{for each adjacent neighbor } \mathbf{x}_{nb} \text{ of } \mathbf{x} \\ \text{do } \begin{cases} \text{if } \mathbf{x}_{nb} \text{ is not in } L \\ \text{then } \begin{cases} p \leftarrow U(\mathbf{x}_{nb}) \\ q \leftarrow \text{solution of } g(\mathbf{x}_{nb}) = 0 \\ \text{if } p > q \\ \text{then } \begin{cases} U(\mathbf{x}_{nb}) \leftarrow q \\ \text{add } \mathbf{x}_{nb} \text{ to } L \end{cases} \end{cases} \end{cases} \end{cases} \\ \text{remove } \mathbf{x} \text{ from } L \end{cases} \end{cases}$

---



---

ALGORITHM 4.3. SOLVE QUADRATIC( $a, b, c, f$ ).

---

**comment:** Returns the value  $u = U(\mathbf{x})$  of the solution of (2) ( $c \leq b \leq a$ )

$u \leftarrow c + 1/f$

**if**  $u \leq b$  **return** ( $u$ )

$u \leftarrow (b + c + \text{sqrt}(-b^2 - c^2 + 2bc + 2/f^2))/2$

**if**  $u \leq a$  **return** ( $u$ )

$u \leftarrow (2(a + b + c) + \text{sqrt}(4(a + b + c)^2 - 12(a^2 + b^2 + c^2 - 1/f^2)))/6$

**return** ( $u$ )

---

**4.2. Properties of the algorithm.** In this section we describe how the algorithm works in detail. Figure 3 shows the schematic 2D example of the FIM wavefront expanding in the first quadrant. The lower-left corner grid node is the source node, the black nodes are fixed nodes, the blue nodes are the active nodes, the diagonal rectangle containing active node is the active list, and the black arrow represents the

narrow band's advancing direction. Figure 3(a) is the initial stage, (b) is after the first update step, and (c) is after the second update step. Because blue nodes depend only on their adjacent neighbor nodes (black nodes), all of the blue nodes in the active list can be updated concurrently. If the characteristic path does not change the direction to the other quadrant, then all the updated blue nodes can be fixed (i.e., become black nodes) and their white adjacent nodes will form a new narrow band.

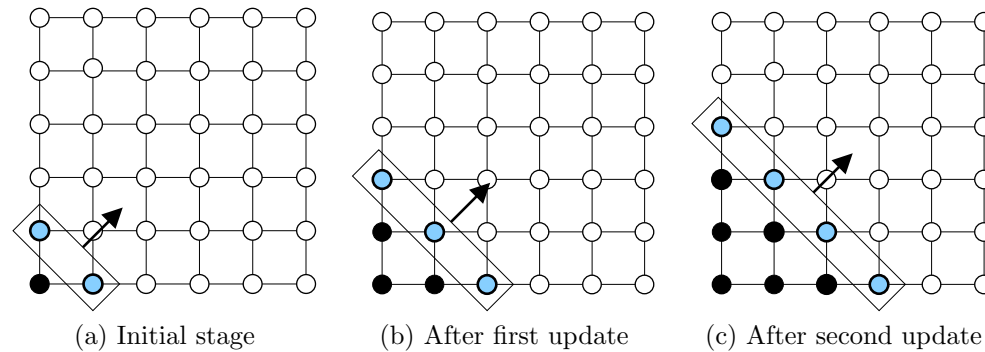


FIG. 3. Schematic 2D example of FIM wavefront propagation.

FIM is an iterative method, meaning that a node is updated until it converges. However, for many datasets, most nodes require only a single update to converge. This can be interpreted as follows. If the angle between the direction of the characteristic path and the narrow band's advancing direction is less than 45 degrees, the exact solution at the node can be found only with a single update, as with FSM. If the angle is larger than 45 degrees, the nodes at the location where the characteristic path changes the direction will have initial values that are computed using wrong upwind neighbor nodes, and they will be revised in successive iterations as neighbors refine their values. Thus, such nodes will not be removed from the active list and will be updated until the correct solutions are finally computed. Figure 4 shows such a case. Unlike FMM, where the wavefront propagates with closed, 1-point-thick curves, FIM can result in thicker bands that split in places where the characteristic path changes the direction (Figure 4(a) red point). Also, the wavefront can move over solutions that have already converged and reactivate them to correct values as new information is propagated across the image. Note that the sweeping in the context of FIM is not the same as in FSM. Usually the shape of the narrow band (active list) is not straight lines as depicted in Figure 3 unless the source is straight lines or a single point and the speed contrast is not high. Some regions might have very complex characteristic direction changes, and on these regions not lines but a group of points will be formed as the active list and advanced altogether until all of them disappear by converging to correct solutions.

In this paper we compute viscosity solutions of the Eikonal equation and rely on a Godunov upwind discretization of the Hamiltonian, as is done in the literature [28, 30, 40]. A characterization of the numerical convergence of the discrete grid to the continuous viscosity solution is given in these other works and is beyond the scope of this paper. The proposed computational algorithm is based on a numerical consistency of the proposed method with these discrete approximations. That is, for a given set of boundary conditions and a particular numerical scheme, there is a unique (to within representational error) discrete approximation on the grid. We show

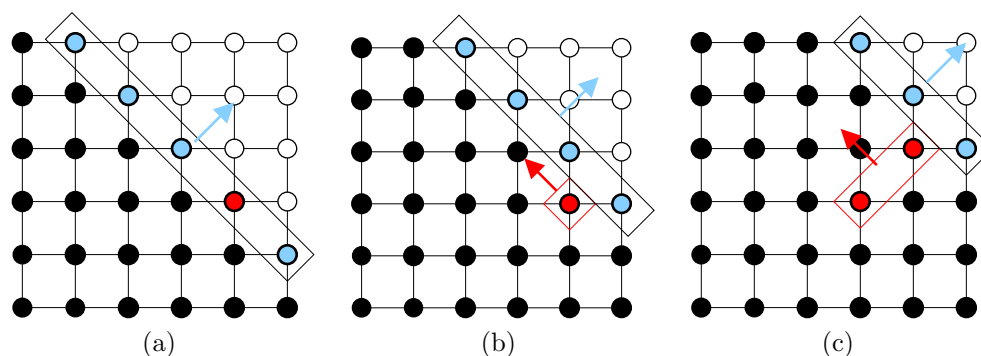


FIG. 4. Schematic 2D example of the change of the characteristic direction.

that FIM converges to this solution and therefore demonstrates the same numerical properties as these previous algorithms.

To briefly prove correctness of the proposed algorithm, we follow reasoning similar to that described in [28].

LEMMA 4.1. *For strictly positive speed functions, the FIM algorithm appends every grid point to the active list at least once.*

*Proof.* For every nonsource point, any path in the domain from that point to the boundary conditions has cost  $< \infty$ . As shown in the initialization step in pseudocode (Algorithm 4.2), all nonsource points are initialized as  $\infty$ . Hence, the active list grows outward from the boundary condition in one-connected rings until it passes over the entire domain.  $\square$

LEMMA 4.2. *FIM algorithm converges.*

*Proof.* For this we rely on monotonicity (decreasing) of the solution and boundedness (positive). From the pseudocode in Algorithm 4.2 we see that a point is added to the active list and its tentative solution is updated only when the new solution is smaller than the previous one. All updates are positive by construction.  $\square$

LEMMA 4.3. *The solution  $U$  at the completion of the FIM algorithm with  $\epsilon = 0$  (error threshold) is consistent with the corresponding Hamiltonian given in (1).*

*Proof.* Each point in the domain is appended to the active list at least once. Each point  $\mathbf{x}$  is finally removed from  $\mathcal{L}$  only when  $g(U, \mathbf{x}) = 0$  and the upwind neighbors (which impact this calculation) are also inactive. Any change in those neighbors causes  $\mathbf{x}$  to be reappended to the active list. Thus, when the active list is empty (the condition for completion),  $g(U, \mathbf{x}) = 0$  for the entire domain.  $\square$

THEOREM 4.4. *The FIM algorithm for  $\epsilon = 0$  gives an approximate solution to (1) on the discrete grid.*

*Proof.* The proof of the theorem is given by the convergence and consistency of the solution, as given in the lemmas above.  $\square$

**4.3. Extension to SIMD parallel architectures.** The proposed method can be easily implemented using multiple threads on parallel systems, e.g., shared memory multiprocessor systems or multicore processors, by assigning each active node to a thread and updating multiple nodes in parallel. However, modern parallel architectures are equipped with SIMD computation units, and such architectures have special hardware characteristics. First, the architecture is highly efficient on the data parallel streaming computing model. A group of threads executes the same instruction on multiple data streams, and therefore a synchronous update without branching is

efficient. Second, the architecture provides extremely wide memory bandwidth which favors coherent memory access, so updating spatially coherent nodes is preferred. Third, the architecture provides a small, user-configurable, on-chip memory space which is as fast as registers on a conventional CPU. Effective use of this on-chip memory, called shared memory or local storage, requires careful modification to the traditional programming model.

Therefore, we propose a variant of FIM, which we call *block FIM*, that scales well on SIMD architectures, based on the idea of a block-based update scheme, similar to the block-based virtual memory framework introduced in [20]. The main idea is splitting the computational domain (grid) into multiple nonoverlapped blocks and using each block as a computing primitive as a node in the original FIM algorithm. The active list maintains the *active blocks* instead of active nodes, and a whole active block is updated by a SIMD computing unit during each iteration. In each iteration, active blocks are copied to the local memory space, and internal iterations are executed to update each block even faster without reading back and forth from the slow off-chip memory. The pseudocode for the block FIM algorithm is given in Algorithm 4.4.

---

ALGORITHM 4.4. BLOCK FIM( $V, L$ ).

---

```

comment: Update blocks  $\mathbf{b}$  in active list  $L$ ,  $V$ : set of all blocks
while  $L$  is not empty
  {
    comment: Step 1 - Update active blocks
    for each  $\mathbf{b} \in L$  in parallel
      do {
        for  $i = 0$  to  $n$ 
          do {
             $\{(U(\mathbf{b}), C_n(\mathbf{b})) \leftarrow \text{solution of } g(\mathbf{b}) = 0$ 
             $C_b(\mathbf{b}) \leftarrow \text{reduction}(C_n(\mathbf{b}))$ 
          }
        comment: Step 2 - Check neighbors
        for each  $b \in L$  in parallel
          do {
            if  $C_b(b) = \text{TRUE}$ 
              do {
                then {
                  for each adjacent neighbor  $\mathbf{b}_{nb}$  of  $\mathbf{b}$ 
                    do {
                       $\{(U(\mathbf{b}_{nb}), C_n(\mathbf{b}_{nb})) \leftarrow \text{solution of } g(\mathbf{b}_{nb}) = 0$ 
                       $C_b(\mathbf{b}_{nb}) \leftarrow \text{reduction}(C_n(\mathbf{b}_{nb}))$ 
                    }
                }
              }
            comment: Step 3 - Update active list
             $\text{clear}(L)$ 
            for each  $\mathbf{b} \in V$ 
              do {
                if  $C_b(\mathbf{b}) = \text{FALSE}$ 
                  then {insert  $\mathbf{b}$  to  $L$ 

```

---

Algorithm 4.4 is a straightforward extension of Algorithm 4.2 to SIMD parallel architectures except for a few differences in notation and additional operators.  $U(\mathbf{b})$  represents a set of discrete solutions of the Eikonal equation, and  $g(\mathbf{b})$  represents a block of Godunov discretization of the Hamiltonian for a grid block  $\mathbf{b}$ .  $C_n$  and  $C_b$  represent the convergence for nodes and a block for  $\mathbf{b}$ , respectively. For example,  $C_n(\mathbf{b})$  is a set of boolean values where each boolean value represents the convergence of the corresponding node in the block  $\mathbf{b}$ , and  $C_b(\mathbf{b})$  is a single boolean value that

represents the convergence of the whole block  $\mathbf{b}$ . If all nodes in the block  $\mathbf{b}$  converge, then  $C_b(\mathbf{b})$  is true; otherwise, it is false.  $C_n(\mathbf{b})$  is set whenever  $U(\mathbf{b})$  is updated. To compute  $C_b(\mathbf{b})$ , we introduce a new operator *reduction*, which is commonly used in a streaming programming model to reduce a larger input stream to a smaller output stream. Reduction can be implemented using an iterative method using parallel threads. For each iteration, every thread reads two boolean values from the current block and writes back the result of the *AND* operation of two values. The number of the threads to participate in this reduction is halved in the successive iteration, and this is repeated until only one thread is left. By doing this, for a block of size  $n$ , only  $O(\log_2 n)$  computation is required to reduce a block to a single value.

**5. Results and discussion.** Several algorithms from the literature report worst- or best-case performances, but to understand the actual performance of algorithms in realistic settings, we have conducted a systematic empirical comparison. First, we show the result of CPU implementation to discuss the intrinsic characteristics of the existing and proposed methods on a single processor. Then we show the result of GPU implementation to show the performance of the proposed algorithm on parallel architecture. We have constructed five different 3D synthetic speed examples for input to the Eikonal solvers. The size of each volume is  $256^3$ . These speed functions capture various situations that are important to the relative performance of the different solvers including constant speeds, random speeds, and large characteristic direction changes. We tested each solver five times on each example and measured the average running time. The speed examples we use for the experiment are as follows:

*Example 1.*  $f = 1$ , a constant speed.

*Example 2.* Three layers of different speed.

*Example 3.* Correlated random speed.

*Example 4.* Maze-shaped impermeable barriers over a constant speed.

*Example 5.* Maze-shaped permeable barriers (speed = 0.001) over a constant speed.

The boundary condition for Examples 1, 2, and 3 is a single point at the center of the grid with a distance of zero. For Examples 4 and 5, we place a seed point on the bottom left corner of every  $z$  slice with a distance of zero. All the tables given below show the running times in seconds. The relative difference in solutions that constitutes convergence for both FSM and FIM is  $10^{-6}$ .

**5.1. CPU implementation.** We have implemented and tested Eikonal solvers on a Windows XP PC equipped with an Intel Core 2 Duo 2.4 GHz CPU and 4 GBytes of main memory. In this section, we focus only on single processor performance and compare the running time of Eikonal solvers on different speed examples. Table 1 shows the running time of CPU-based Eikonal solvers on Examples 1, 2, and 3.

TABLE 1  
Running time on Examples 1, 2, and 3.

	Example 1	Example 2	Example 3
FMM	65.5	65.0	77.7
GMM	29.1	35.1	38.9
FSM	33.3	94.2	231.4
FIM	23.6	28.2	92.3

Example 1 is the simplest case to compute a distance field over a homogeneous media from a single source point (Figure 5 (a)). The speed is constant ( $f = 1$ ), so the

$t$ -level set of  $\phi$  is the circle with a radius  $t$  from the source. On this example, FSM converged only after nine sweeps due to the simplicity of the speed function. GMM, FSM, and FIM run two to three times faster than FMM on this example. This is an example of the linear complexity algorithm outperforming the worst-case-optimal algorithm (i.e., Dijkstra-type algorithms).

Example 2 is an example of three different speed functions (Figure 5(b)). The speed of the bottom layer is the slowest, and two upper layers are two and three times faster than the bottom layer, respectively. This example may represent, for instance, different rock materials, which are often found in wave propagation examples that arise in geoscience applications. This simple change of the speed function from Example 1 can degrade the performance of the linear complexity algorithms, specifically FSM, due to the extra updates required for speed variations. FSM requires 23 sweeps to converge, which takes a computation time three times longer than that on Example 1. GMM and FIM run slightly more slowly than Example 1; however, FMM is not affected much by the speed changes. FIM still runs fastest among the others, and this shows that when the input data does not have much speed variation, the proposed method can outperform the other methods.

Example 3 is a correlated random speed map generated by blurring and thresholding random noise (Figure 5(c)). This example mimics the speed functions observed in real seismic data. Note that the running time of FMM and GMM did not increase much, but the running time of FSM and FIM increased significantly compared to Examples 1 and 2. Due to the randomness of the speed, the level curves of the solution are curly, and therefore there are many local characteristic direction changes which require many sweeps in FSM (54 sweeps) or multiple relisting of nodes in FIM. Real world data contain noise and varying speeds as in this example, and therefore we may say that the worst-case-optimal method (FMM) can perform better than the linear iterative methods (FSM, FIM) in realistic settings on a single processor.

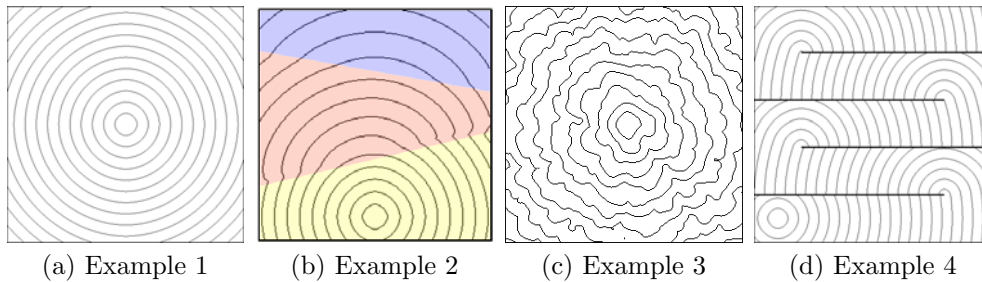


FIG. 5. Level curves of the solution of examples.

Examples 4 and 5 are designed to simulate frequent characteristic direction changes on a maze-like shape domain. This example is designed to show where the performance of FSM and FIM is significantly degraded by forcing the characteristic paths to turn multiple times (inefficient for FSM) or by making Euclidean distance shorter than the actual distance from the solution of the Eikonal equation (inefficient for FIM). Figure 5(d) shows the example with four impermeable barriers, and Table 2 shows the results of two sets of experiments, one with impermeable barriers and the other with permeable barriers but very slow (speed = 0.001). Each experiment set consists of four tests with a different number of barriers, from 2 to 16, on a  $256^3$  input volume with a constant speed  $f = 1$ , and a set of source points is placed at the lower left corner.

TABLE 2  
*Running time on speed—Examples 4 and 5.*

	Example 4				Example 5			
	2	4	8	16	2	4	8	16
FMM	29.9	28.8	26.8	24.2	31.6	31.5	30.6	29.8
GMM	22.7	21.4	20.0	17.6	30.1	36.8	51.6	80.2
FSM	22.5	37.0	70.6	140.3	22.8	38.0	67.0	126.0
FIM	18.7	18.5	17.6	16.3	36.1	43.2	62.9	106.4

In this experiment, we find several interesting characteristics of the methods. First, FMM is not much affected by the number of barriers or permeability of the barrier, which clearly demonstrates the advantage of the worst-case-optimal method. FMM runs even faster with multiple barriers because the wavefront is shorter as the number of barriers increases (because the pathway where the wave passes is narrower) and therefore the depth of the heap becomes shallower and the method runs faster. Second, GMM runs as fast as FMM with impermeable barriers but runs slowly with permeable barriers. This is because the speed on the barrier is very slow ( $f = 0.001$ ) compared to the off-barrier region ( $f = 1$ ) in Example 5, and due to the implementation scheme we used [18], the method wastes much time on searching a group of points to update using a global threshold with a small increment on each iteration. Third, FSM runs very slowly on these examples because the wave should turn multiple times at the end of each barrier, and for each set of sweeps (8 sweeps for three dimensions) only a piece of the region where the wave can reach with less than a 180 degree change of the characteristics can be updated correctly. Fourth, the performance of FIM does not depend on how many times the characteristic curve turns but depends on how many times nodes are updated to converge to the correct solutions. For Example 4, the wave from the source point cannot penetrate the barriers, and therefore every node can compute the correct solution with only a single wave propagation even though the wave should turn multiple times along the maze. However, for Example 5, the initial wave from the source will penetrate the barriers and the nodes behind the barriers will get incorrect solutions, and therefore there should be additional updates using follow-up waves created at the end of each barrier. This happens because the motion of the wavefront in FIM is not governed by the solution (or the speed function) of the Eikonal equation as in FMM but the Euclidean distance (or grid distance) from the source. Therefore, the performance of FIM on Example 5 degrades significantly proportional to the number of barriers. Figure 6 shows several steps of the FIM wave propagation on the maze data with permeable barriers. The source points are placed on the lower left corner of the image, and yellow lines represent the active list (wavefront). Blue to red color indicates the distance from the source point. Note that after the first wave sweeps from lower left to upper right (Figure 6, two images from the left), several follow-up waves are created and sweep the image again to revise the incorrect solutions (Figure 6, middle to right images).

**5.2. GPU implementation.** To show the performance of FIM on SIMD parallel architectures, we have implemented and tested block FIM (Algorithm 4.4) on an NVIDIA Quadro FX 5600 graphics card using NVIDIA CUDA [23]. The NVIDIA Quadro FX 5600 graphics card is equipped with 1.5 GBytes memory and 16 microprocessors, where each microprocessor consists of eight SIMD computing cores that run in 1.35 GHz. Each computing core has 16 KBytes of on-chip shared memory for faster access to local data. 128 cores physically run in parallel, but the number of

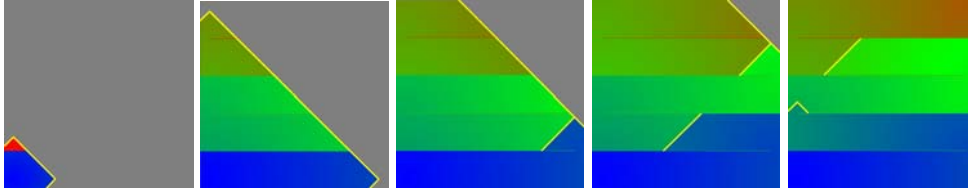


FIG. 6. Wave propagation on maze data with four permeable barriers (Example 5).

threads running on a GPU should be much larger because cores are time-shared by multiple threads to maximize the throughput and increase computational intensity. Computation on the GPU entails running a kernel with a batch process of a large group of fixed size thread blocks, that maps well to the block FIM algorithm that uses block-based update methods. We fix the 3D block size to  $4^3$  in our implementation to balance the GPU resource usage (e.g., registers and shared memory) and the number of threads running in parallel.

Table 3 shows the running time of the block FIM method on the GPU on the same input volumes ( $256^3$ ) used to test the CPU Eikonal solvers in section 5.1. Examples 4 and 5 are maze datasets with 16 (im)permeable barriers, respectively. The numbers in the brackets represent the measured speed-up factor of block FIM (on the GPU) over FMM (on the CPU).

TABLE 3  
Running time of block FIM on  $256^3$  volumes and measured speed-up compared to FMM.

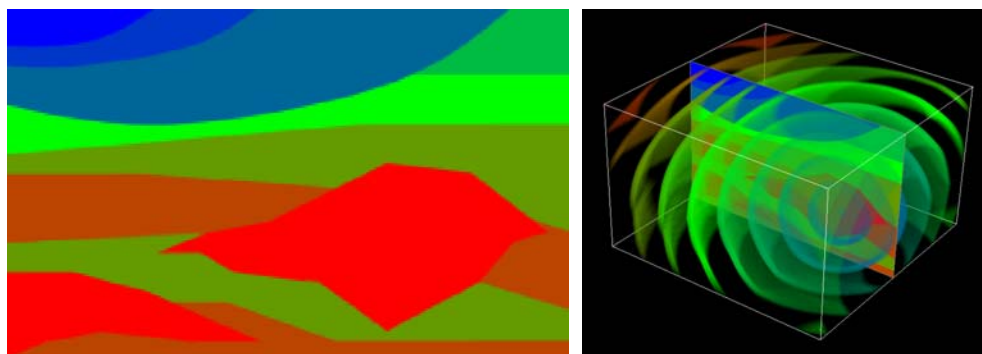
Example 1	Example 2	Example 3	Example 4	Example 5
0.55 (119x)	1.01 (64x)	1.81 (43x)	1.41 (17x)	4.88 (6x)

The proposed block FIM algorithm maps very well to the GPU and achieves a huge performance gain over the traditional CPU-based solvers. On a simple case such as Example 1, block FIM runs about 120 times faster than FMM, which is about 30 million distance computations per second. On more complex cases such as Examples 2 and 3, block FIM still runs about 40–60 times faster than FMM. On the most challenging case, Example 5, where FIM runs roughly four times slower than FMM on the CPU, block FIM still runs about six times faster than FMM. In summary, block FIM implemented on the GPU runs faster than any existing CPU-based solver on all examples we tested, and many time-consuming Eikonal equation applications can run at real-time or interactive rates using the proposed method, as shown in the following section.

**5.3. Applications.** In this section we show two applications of Eikonal equation solvers. The first application is the seismic travel time tomography. Figure 7 (left) shows a slice of the synthetic speed volume where different colors indicate different wave propagation speeds (red is the fastest and blue is the slowest speed), and the right figure shows the level surfaces of the solution of the Eikonal solver. Once a speed map is given, then we can simulate the wave propagation by solving the Eikonal equation with user-defined boundary conditions.

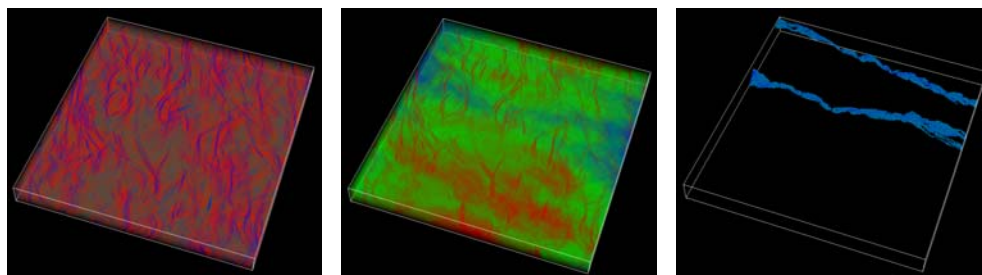
In this application, users can interactively pick source points on a slice of the speed volume, and then the Eikonal solver calculates the solution of the equation and displays the 3D level surfaces. We have tested both FMM and block FIM to compare the single and the parallel solver performances. For a  $256 \times 256 \times 128$  input volume,



FIG. 7. *Seismic tomography on synthetic speed volume.*

block FIM implemented on the GPU runs in less than a second, so the user can interactively explore the seismic volume by placing a source point on many different locations and get results in real time. However, FMM (on the CPU) runs slowly, taking roughly one minute to compute a solution on this dataset, and therefore it is not suitable for interactive tomography applications.

The second application is the connectivity analysis in the clay-plastic compound tire dataset (data courtesy of ExxonMobil Chemical Europe, Inc., and Upstream Research Company). Figure 8 left shows the molecular microstructure of a part of a clay-ethylene vinyl acetate tire compound dataset reconstructed from an electron tomography scan of the object. The goal of this application is to analyze the quality of the molecular-level connectivity in the data using a two-way propagation scheme based on the Hamilton–Jacobi framework, as introduced in [16]. To quantify the connectivity, first we assign a proper speed value per voxel according to its material type (Figure 8 left, blue to red, slow to fast), and then the user assigns two end regions of interest to compute connectivity. Next, we run the Eikonal solver twice, each time with one of the end regions as the boundary condition of the equation, and we combine the two solutions together to finally extract the minimum cost paths between the regions. Figure 8 middle shows the connectivity between two end x slices, where blue indicates strong connectivity and red indicates weak connectivity. Figure 8 right shows the volumetric paths between the end slices that have strong connectivity. Block FIM runs in less than 15 seconds for each Eikonal equation on a GPU, while FMM takes more than 5 minutes to do the same calculation on a CPU.

FIG. 8. *Connectivity analysis on a clay-plastic compound tire volume.*

**5.4. Discussion.** In the previous sections, performances of the Eikonal solvers are compared based on the actual running times on a single CPU and a GPU. Because running time can be affected by many factors, e.g., implementation schemes or hardware performance, we measure the number of computations for a more careful performance analysis. The most time consuming operation for the Eikonal equation solver is computing the solution of quadratic equations. Table 4 compares the average number of solving quadratic equations per node on the examples.

TABLE 4  
*Average update operations per node on  $256^3$ .*

	Example 1	Example 2	Example 3	Example 4	Example 5
FMM	2.9	2.9	2.9	1.9	1.9
GMM	5.8	4.4	4.5	5.1	4.9
FSM	9	23	54	34	34
FIM	4.9	5.2	10	4.0	24

For FMM, the tentative solutions on the wavefront must be computed multiple times, and each node has six neighbors on a 3D grid; therefore, we can assume that about half of the adjacent neighbors are fixed when a node is updated. Thus, each node is updated roughly three times on average. On Examples 4 and 5, we use a set of source points that forms a line along the  $z$  direction; therefore, the wave propagation is only along the  $x$  and  $y$  directions and the average computation per node is about two. For GMM, each node in the group is updated twice for stability (i.e., alternating one-dimensional (1D) Gauss–Seidel), so the average number of computations is about two times that of FMM, which is six, though a heuristic to skip the second update slightly reduces the average update numbers. The average number of computations is same as the number of sweeps for FSM, and it highly depends on the speed settings. FIM also depends on the speed settings, but due to its local operations, the average number of updates is not as large as that for FSM. On most examples, FIM requires less than ten updates per node on average. On a special configuration like Example 5, FIM requires a large number of operations but still is not as expensive as FSM.

Even though FMM's average number of updates is much smaller than that of FIM, the actual running time is longer than that of FIM on some examples. The reason is because managing a heap in FMM can be expensive, especially three dimensions. Let  $k_1$  and  $k_2$  be the cost for a quadratic solver computation and a heap updating operation, respectively, and  $P_{FMM}$  and  $P_{FIM}$  are the average number of operations per node in FMM and FIM, respectively (as in Table 4). Let  $h$  be the average heap size. The total cost for FMM and FIM on the grid size  $N$  can be defined asymptotically as follows:

$$C_{FMM} = N(k_1 P_{FMM} + k_2 P_{FMM} \log_2(h))$$

$$= N k_1 P_{FMM} \left( 1 + \frac{k_2}{k_1} \log_2(h) \right)$$

$$C_{FIM} = N k_1 P_{FIM}.$$

Hence, for FIM to outperform FMM, the following condition must hold:

$$Nk_1P_{\text{FMM}} \left( 1 + \frac{k_2}{k_1} \log_2(h) \right) > Nk_1P_{\text{FIM}},$$

$$P_{\text{FIM}} < P_{\text{FMM}} \left( 1 + \frac{k_2}{k_1} \log_2(h) \right).$$

For example, empirically measured  $\frac{k_2}{k_1}$  is about 0.08 and the average heap size  $h$  (the number of elements in the heap) is 104857 for FMM on Example 1. Therefore,  $\frac{k_2}{k_1} \log_2(h) \approx 1.3$ , and because  $P_{\text{FMM}}$  is 2.9 (Table 4), if  $P_{\text{FIM}} < 6.75$ , then FIM may outperform FMM on a single processor system. If we assume that the average heap size does not significantly vary on the examples we tested, this reasoning also matches with our experiment results because FIM outperforms FMM on Examples 1, 2, and 4 whose  $P_{\text{FIM}}$  are smaller than 6.75 (for Example 4,  $\log_2(h)$  is smaller than the other examples, but  $P_{\text{FIM}}$  is also smaller). We can derive a similar analysis for block FIM by defining the total cost for block FIM as  $C_{\text{MFIM}} = \frac{N}{M}k_1P_{\text{FIM}}$ , where  $M$  is a parallel speed-up factor for a specific parallel hardware. Even though it is not easy to exactly define  $M$  because it depends not only on the number of parallel computing units but also on the other hardware characteristics, e.g., size of caches, memory bandwidth, based on our experiment results on the GPU,  $M$  could be roughly 50 on the graphics card we used.

The total cost for FMM depends not only on the size of the input but also on the average size of the heap (or area of level surfaces), and therefore the performance of FMM decreases more rapidly than that of the other methods. Figure 9 compares the running time of CPU-based Eikonal solvers on various size volumes to show how the data size affects the performance of each solver. In this experiment, we fix the  $x$  and  $y$  dimension to 512 but increase the  $z$  dimension from 10 to 100. The speed is constant over the whole volume. We placed the seed points at the center of each  $z$  slice; therefore, the seed points form a line in three dimensions and its wavefront propagates as a cylindrical shape. In this setup, the surface area of the wavefront grows proportional to the size of the data. In Figure 9, the slope of the running time for FMM is much steeper than that of the other methods, and this shows that the heap becomes a bottleneck for FMM as the size of the wavefront increases.

In block FIM, every active block is updated multiple times before its convergence is checked for two reasons: (1) not all the nodes in a block can converge only with a single update, and (2) the fast on-chip shared memory space can serve as a scratch area for iterative computations without communicating with the main memory. Therefore, the iteration number  $n$  per block should be chosen as a user-defined parameter. If  $n$  is too small, then the method checks the convergence of a block before it is actually converged, and if  $n$  is too big, then it executes unnecessary extra updates after convergence. If there are no characteristic direction changes within a block, then theoretically a  $k^3$  block can converge using at most  $3k - 2$  updates (in our examples,  $k = 4$ , and therefore  $n = 10$ ), which is the number of steps required to sweep through from one corner to its diagonal corner of the block. However, the number depends not only on the size of the block but also on the input speed function. Figure 10 compares the running time of block FIM on the GPU using various iteration numbers per block. The running time for  $n < 10$  is not stable and has a few spikes, but for  $n > 10$  the running time becomes stable and gradually increases as  $n$  increases. This is because a block usually converges with 10 or more updates, and therefore wavefront

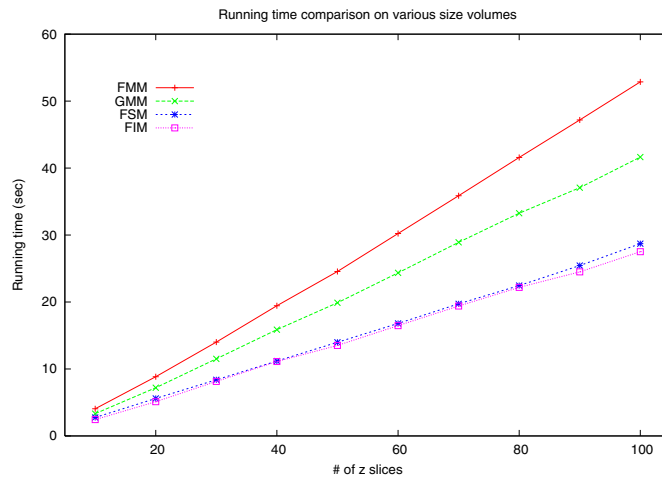


FIG. 9. Running time comparison on volumes of various size.

propagation is almost identical with  $n > 10$  iterations. In the same manner, we can argue that  $n < 10$  is not enough for some blocks to converge, and therefore such an early convergence check induces updating blocks in different sequences, which leads to different wavefront propagation. Note that, in contrast to the other examples, the running time of Example 3 is shorter for smaller  $n$ , and this is due to the randomness of the speed functions. Example 3 is created using random speed values, which causes locally varying characteristic directions (that is why the level curves of Example 3 are curly), and for such datasets fewer iterations per block can help to propagate the correct information across blocks in the early stage and reduce computations using incorrect upwind neighbors. In general, according to our experiments, the best  $n$  can be around 12 for most cases and around 5 for highly random speed values for a  $4^3$  block.

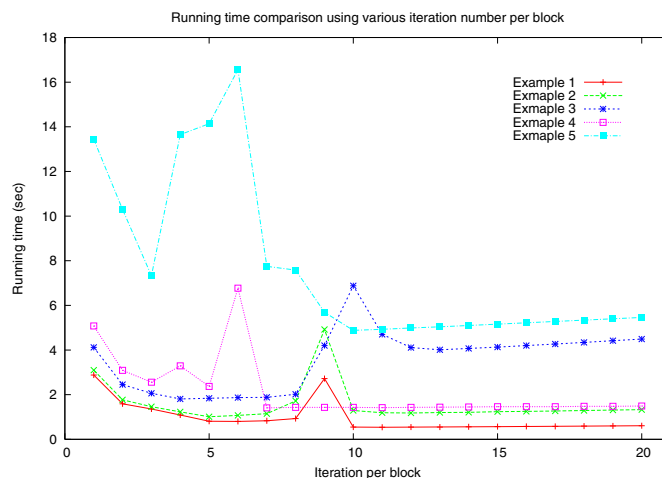


FIG. 10. Running time comparison using various iterations per block.

In summary, we have compared the existing Eikonal solvers with the proposed

method and found several interesting characteristics of the methods. FMM works reasonably well on most cases, and the running time is not affected by the input speed functions. However, the bottleneck of FMM is the heap, so the performance will degrade as the data size and dimension become larger, and parallelization is difficult. Surprisingly, GMM works well and faster than FMM on most cases unless the speed contrast is too high, so it can be a good substitute for FMM on larger datasets. FSM did not perform well on most cases except a very simple case such as Example 1, and it is in agreement with the result of [14]. Both GSM and FSM may be implemented on multiprocessor parallel systems [41], but the implementation might be limited to non-SIMD architectures. The proposed method, FIM, works reasonably well even on a single processor for simple speed settings, such as Examples 1 and 2, and we have given a brief theoretical analysis on this result above. However, the results of Examples 4 and 5 also show that FIM has a weakness on complex datasets. FIM maps well on any parallel architectures, and we have learned that block FIM implemented on the GPU outperforms all the examples we tested because the penalty incurred by the suboptimal algorithm can be compensated for by the parallelism of SIMD architectures. Table 5 summarizes the performance of the discussed Eikonal solvers over several data categories, where + implies the best,  $\triangle$  is so-so, and  $-$  is the worst.

TABLE 5  
*Overall performance comparison on data categories.*

	FMM	GMM	FSM	FIM
Complex speed setup	+	+	-	-
Large data size	-	+	+	+
Parallelization	-	$\triangle$	$\triangle$	+

**6. Conclusion and future work.** In this paper we propose a fast algorithm to solve the Eikonal equation on parallel architectures. The proposed algorithm is based on a label-correcting method with a modification for better fitting to SIMD parallel architectures. The method employs a narrow band method to keep track of the grid nodes to be updated and iteratively updates nodes until they converge. Instead of using an expensive sorting data structure to keep the causality, the proposed method uses a simple list to store active nodes and updates all of them concurrently using a Jacobi update scheme. The nodes in the list can be removed from or added to the list based on the convergence measure. The method is simple to implement and runs faster than the existing methods on a wide range of speed inputs. The method is also easily portable to parallel architectures, which is difficult or not available on many existing methods. We compared the performance of the proposed method with existing methods on both a single processor and an SIMD parallel processor.

The proposed method introduces numerous interesting future research directions. Many time-consuming applications using the Eikonal equation solver can benefit from the proposed FIM method on parallel architectures, and we are looking for time-critical interactive applications of the Eikonal equation in the field of geoscience or medical image analysis. In addition, it will be interesting to develop a new numerical scheme, other than the Godunov finite difference method, that fits better on SIMD architectures to solve the general Hamilton–Jacobi equation. Applying FIM to solve other boundary value PDE problems would be another interesting future work.

## REFERENCES

- [1] D. BERTSEKAS, F. GUERRIERO, AND R. MUSMANNO, *Parallel asynchronous label correcting methods for shortest paths*, J. Optim. Theory Appl., 88 (1996), pp. 297–320.
- [2] D. P. BERTSEKAS, *Dynamic Programming and Optimal Control*, Vol. 1, 3rd ed., Athena Scientific, Nashua, NH, 2005.
- [3] F. BORNEMANN AND C. RASCH, *Finite-element discretization of static Hamilton-Jacobi equations based on a local variational principle*, Comput. Vis. Sci., 9 (2006), pp. 57–69.
- [4] A. M. BRONSTEIN, M. M. BRONSTEIN, Y. S. DEVIR, R. KIMMEL, AND O. WEBER, *Parallel algorithms for approximation of distance maps on parametric surfaces*, Technical report CIS-2007-03, Technion, Haifa, Israel, 2007.
- [5] A. M. BRONSTEIN, M. M. BRONSTEIN, AND R. KIMMEL, *Weighted distance maps computation on parametric three-dimensional manifolds*, J. Comput. Phys., 225 (2007), pp. 771–784.
- [6] A. BRUSS, *The eikonal equation: Some results applicable to computer vision*, J. Math. Phys., 23 (1982), pp. 890–896.
- [7] I. BUCK, T. FOLEY, D. HORN, J. SUGERMAN, K. FATAHALIAN, M. HOUSTON, AND P. HANRAHAN, *Brook for GPUs: Stream computing on graphics hardware*, ACM Trans. Graph., 23 (2004), pp. 777–786.
- [8] CELL BROADBAND ENGINE RESOURCE CENTER, <http://www.ibm.com/developerworks/power/cell/>.
- [9] P.-E. DANIELSSON, *Euclidean distance mapping*, Comput. Graphics and Image Processing, 14 (1980), pp. 227–248.
- [10] M. FALCONE, *A numerical approach to the infinite horizon problem of deterministic control theory*, Appl. Math. Optim., 15 (1987), pp. 1–13.
- [11] M. FALCONE, *The minimum time problem and its applications to front propagation*, in Motion by Mean Curvature and Related Topics (Trento, 1992), de Gruyter, Berlin, 1994, pp. 70–88.
- [12] M. FALCONE AND P. LANUCARA, *Parallel algorithms for Hamilton-Jacobi equations*, in ICIAM/GAMMJ95 Special Issue on Zeitschrift für Angewandte Mathematik und Mechanik (ZAMM), Appl. Stochastics Optim. 3, O. Mahrenholtz, K. Marti, and R. Mennicken, eds., Akademie Verlag, Berlin, 1996, pp. 355–359.
- [13] I. FISCHER AND C. GOTSMAN, *Fast Approximation of High Order Voronoi Diagrams and Distance Transforms on the GPU*, Technical report CS TR-07-05, Harvard University, Cambridge, MA, 2005.
- [14] P. A. GREMAUD AND C. M. KUSTER, *Computational study of fast methods for the eikonal equation*, SIAM J. Sci. Comput., 27 (2006), pp. 1803–1816.
- [15] K. E. HOFF III, J. KEYSER, M. LIN, D. MANOCHA, AND T. CULVER, *Fast computation of generalized Voronoi diagrams using graphics hardware*, in Proceedings of SIGGRAPH 1999, ACM, New York, 1999, pp. 277–286.
- [16] W.-K. JEONG, P. T. FLETCHER, R. TAO, AND R. WHITAKER, *Interactive visualization of volumetric white matter connectivity in DT-MRI using a parallel-hardware Hamilton-Jacobi solver*, IEEE Trans. Visualization and Computer Graphics, 13 (2007), pp. 1480–1487.
- [17] U. KAPASI, W. J. DALLY, S. RIXNER, J. D. OWENS, AND B. KHAILANY, *The Imagine stream processor*, in Proceedings of the 2002 IEEE International Conference on Computer Design, IEEE Computer Society, Washington, DC, 2002, pp. 282–288.
- [18] S. KIM, *An  $O(N)$  level set method for eikonal equations*, SIAM J. Sci. Comput., 22 (2001), pp. 2178–2193.
- [19] R. KIMMEL AND J. SETHIAN, *Computing geodesic paths on manifolds*, Proc. Nat. Acad. Sci. USA, 95 (1998), pp. 8431–8435.
- [20] A. LEFOHN, J. KNISS, C. HANSEN, AND R. WHITAKER, *Interactive deformation and visualization of level set surfaces using graphics hardware*, in Proceedings of the IEEE Visualization Conference, IEEE, Washington, DC, 2003, pp. 75–82.
- [21] R. MALLADI AND J. SETHIAN, *A unified approach to noise removal, image enhancement, and shape recovery*, IEEE Trans. Image Processing, 5 (1996), pp. 1554–1568.
- [22] J. MITCHELL, D. MOUNT, AND C. PAPADIMITRIOU, *The discrete geodesic problem*, SIAM J. Comput., 16 (1987), pp. 647–668.
- [23] NVIDIA, *CUDA Programming Guide*, <http://developer.nvidia.com/object/cuda.html>, 2007.
- [24] L. C. POLYMERAKOS, D. P. BERTSEKAS, AND J. N. TSITSIKLIS, *Implementation of efficient algorithms for globally optimal trajectories*, IEEE Trans. Automat. Control, 43 (1998), pp. 278–283.
- [25] F. QIN, Y. LUO, K. OLSEN, W. CAI, AND G. SCHUSTER, *Finite-difference solution of the eikonal equation along expanding wavefronts*, Geophysics, 57 (1992), pp. 478–487.
- [26] N. RAWLINSON AND M. SAMBRIDGE, *The fast marching method: An effective tool for tomo-*

- graphics imaging and tracking multiple phases in complex layered media*, Exploration Geophysics, 36 (2005), pp. 341–350.
- [27] G. RONG AND T.-S. TAN, *Jump flooding in GPU with applications to Voronoi diagram and distance transform*, in Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games, ACM, New York, 2006, pp. 109–116.
  - [28] E. ROUY AND A. TOURIN, *A viscosity solutions approach to shape-from-shading*, SIAM J. Numer. Anal., 29 (1992), pp. 867–884.
  - [29] J. SETHIAN, *A fast marching level set method for monotonically advancing fronts*, Proc. Nat. Acad. Sci. USA, 93 (1996), pp. 1591–1595.
  - [30] J. SETHIAN, *Fast marching methods*, SIAM Rev., 41 (1999), pp. 199–235.
  - [31] J. SETHIAN, *Level Set Methods and Fast Marching Methods*, Cambridge University Press, Cambridge, UK, 2002.
  - [32] J. A. SETHIAN AND A. VLADIMIRSKY, *Ordered upwind methods for static Hamilton–Jacobi equations: Theory and algorithms*, SIAM J. Numer. Anal., 41 (2003), pp. 325–363.
  - [33] R. SHERIFF AND L. GELDART, *Exploration Seismology*, Cambridge University Press, Cambridge, UK, 1995.
  - [34] K. SIDDIQI, S. BOUIX, A. TANNENBAUM, AND S. ZUCKER, *The Hamilton–Jacobi skeleton*, in Proceedings of the IEEE International Conference on Computer Vision, IEEE Computer Society, Washington, DC, 1999, pp. 828–834.
  - [35] C. SIGG, R. PEIKERT, AND M. GROSS, *Signed distance transform using graphics hardware*, in Proceedings of the 2003 IEEE Visualization Conference, IEEE Computer Society, Washington, DC, 2003, pp. 83–90.
  - [36] A. SPIRA AND R. KIMMEL, *An efficient solution to the eikonal equation on parametric manifolds*, Interfaces Free Bound., 6 (2004), pp. 315–327.
  - [37] A. SUD, M. A. OTADUY, AND D. MANOCHA, *Difi: Fast 3d distance field computation using graphics hardware*, Computer Graphics Forum, 23 (2004), pp. 557–566.
  - [38] Y.-H. R. TSAI, L.-T. CHENG, S. OSHER, AND H.-K. ZHAO, *Fast sweeping algorithms for a class of Hamilton–Jacobi equations*, SIAM J. Numer. Anal., 41 (2003), pp. 673–694.
  - [39] J. TSITSIKLIS, *Efficient algorithms for globally optimal trajectories*, IEEE Trans. Automat. Control, 40 (1995), pp. 1528–1538.
  - [40] H. ZHAO, *A fast sweeping method for eikonal equations*, Math. Comput., 74 (2004), pp. 603–627.
  - [41] H. ZHAO, *Parallel implementations of the fast sweeping method*, J. Comput. Math., 25 (2007), pp. 421–429.